INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 5511

Rapports de Recherche

N° 874

A TYPE SYSTEM FOR A DECLARATIVE LANGUAGE

Richard B. KIEBURTZ

JUILLET 1988

RR-0874

UN SYSTEME DE TYPES POUR UN LANGAGE DECLARATIF

A type system for a declarative language

Richard B. Kieburtz
*INRIA - Sophia Antipolis**

28 June 1988

Abstract

TYPOL is a declarative programming language that implements Natural Semantics. A program in this language consists of sets of inference rules that express relations over a syntactic domain of terms generated by an abstract grammar. We define a type system for this language that inherits its types from the phylla of the abstract grammar, and also permits the explicit definition of new types. It incorporates a notion of type hierarchy, or subtypes. Rules for type inference are given in the Natural Semantics formalism.

Résumé

TYPOL est un language programmation déclaratif pour implémenter la Sémantique Naturelle. *Un programme de ce langage consiste en un ensemble de règles d'inférence qui décrivent des relations sur le domaine syntaxique des termes engendrés par une grammaire abstraite. Nous donnons un système de type pour ce langage, dont les types proviennent d'une grammaire abstraite ou sont explicitement définis dans la programme lui-même. Ce système contient une idée de hiérarchie des types, ou sous-types. Les règles d' inférence de type sont présentées dans le formalisme de* Sémantique Naturelle.

*Author's permanent address: Oregon Graduate Center, 19600 N.W. von Neumann Dr., Beaverton, OR 97006, U.S.A.

1 Introduction

Declarative languages enable rapid construction of system protoypes by suppressing many details of evaluation from the system description. Nevertheless, a declarative language is a fully formal logical system, intolerant of imprecise or erroneous statements. It can be a demanding task to capture an intuitive specification accurately and correctly in a declarative language. The task can be made far less tiresome if the language provides a well-defined notion of which of its syntactically well-formed statements are possibly meaningful. It is even better if this notion can be checked automatically. This is the role of a type system in a programming language.

Historically, declarative languages have been formulated without type systems for two reasons. First, language designers had not presumed to know which declarative statements might be meaningful and which were not. Second, until substantial experience had been gained with the use of powerful, declarative languages, few people appreciated fully the need for help in diagnosing misstatements in a large specification.

With increasing use, the need for a type system for a declarative language becomes apparent. In the present paper, we are concerned with one particular declarative language, TYPOL[3]. This is a first-order logic language particularly adapted to prototyping systems that can themselves be defined as programming languages. The definition of TYPOL is given in its own formalism, for instance. TYPOL is a key component of the experimental, interactive environment, Centaur[1], which is being developed as a part of Esprit project GIPE.

TYPOL is a language in which to express rules of a sequent calculus. Thus, given its own inference rules it can be considered to be a mathematical logic. It is also a programming language because its inference rules have been mechanized. A programmed set of rules expressed in this language can be applied to an initial hypothesis to compute elements of a consequent relation. TYPOL's implementation is accomplished by translating its rules into clauses of Prolog. Furthermore, TYPOL has an interface with a grammar-specification language, METAL, which allows it to inherit a multi-sorted algebra from the specification of a grammar for abstract syntax.

A multi-sorted algebra entails a natural type system for expressions of the algebra embedded in the TYPOL language. A TYPOL program can be augmented with programmer-defined types auxiliary to those of the derived algebra, but such auxiliary types are never comparable with the natural types of the inherited algebra. We give a type system for TYPOL that

exploits the natural types of an inherited, multi-sorted algebra and also allows polymorphism with respect to zero-order (i.e. non-functional) types.

On the basis of experience in programming with typed, functional languages we prefer a system in which the types of variables, terms and rules need not be declared but can be inferred. Type inference is much easier to use in practice than is explicit type declaration. During the development of a program, declarations frequently have to be modified to maintain consistency with the use of identifiers in the body of the program. Editing declarations as well as the program body can be tedious.

Type inference eliminates the need for almost all declarations, with two exceptions:

- constructors must be declared in the definition of a new, algebraic type, and
- declarations are required when rule sets are imported from separately compiled modules.

Import declarations can usually be gotten as textual output from previous type inference on the modules to be imported, however.

Another observation favoring type inference is that inferred types are always accurate, whereas those declared by programmers tend either to be overly specific or are too general and break the security of the type system.

The TYPOL type system proposed here uses an extension of Hindley-Milner type inference[9,7]. In principle, this makes it unnecessary to declare types for variables or expressions. It is also consistent with the spirit of modular specification that is manifested in TYPOL by named rule sets. A name can be given to a set of rules related in meaning or associated with a part of a specification. There is no formal basis for the decomposition of an entire specification into modules; this is left to the intuition of the programmer. A program can be checked for type consistency across module boundaries.

A named rule set defines one or more relations over the otherwise free algebra of TYPOL terms. Each such relation is specified by one or more rules, a subset of the named collection of rules. Type inference on a named rule set identifies the relations it specifies by giving a single type to each of them. Inferring types for relations is the counterpart of type inference on function declarations in ML[1]. As in ML, type inference produces a principal

[1]This is by no means the first extension of type inference to a relational language. Type inference has been proposed for Prolog [11] and for Prolog with a notion of subtypes [4]

type for a relation[2]. The type system is only defined for 'pure' TYPOL. It does not account for conditional clauses that are used to impose semantic constraints on the admissibility of rules.

2 The natural types of a multi-sorted algebra

An abstract grammar is the specification of a multi-sorted, freely generated signature algebra, possibly extended by embedding auxiliary algebras such as `ID` (identifiers) or `INT` (integers). Although commonly called a signature (or Σ) algebra, such a specification actually defines a category of algebras. The language defined by an abstract grammar is the carrier set of an initial algebra in such a category.

The finite set of sorts of the algebra corresponds exactly to the set of syntactic phylla of the abstract grammar. To each operator of the grammar, there corresponds a constructor of the algebra whose type is either a simple sort (for a nullary constructor) or a compound type constructed with the mapping operator $\rightarrow$. The form of a compound type is severely restricted: it is either $\zeta_1 \rightarrow \zeta$ or $(\zeta_1 \times \ldots \times \zeta_n) \rightarrow \zeta$, where $\zeta_1, \ldots \zeta_n, \zeta$ are simple sorts. When an operator has a type ζ or a type $\ldots \rightarrow \zeta$, we say that the operator is a constructor for the sort ζ.

Conventionally, the constructors of different sorts are required to form disjoint sets. When an algebra is inherited from a grammar specification, however, the sorts will not necessarily have disjoint sets of constructors. To account for overlapping sets of constructors, we can impose a partial order (called the *subsort relation*) on the sorts, defined in terms of set inclusion of the corresponding constructor sets[3],

$$\zeta_1 \preceq \zeta_2 \;\Leftrightarrow\; Constr(\zeta_1) \subseteq Constr(\zeta_2)$$

It is important that the sorts to which a constructor belongs can always be unambiguously resolved into a least sort. When there is a constructor C belonging to several sorts and the set of sorts to which it belongs, $\mathcal{Z}_C = \{\zeta_i | C \in Constr(\zeta_i)\}$, has no least member in the ordering $\preceq$, then we synthesize a new sort ζ' with

[2]For languages with higher-order relations, subtypes raise problems with the notion of principal type [5]. Since we consider only first-order logics, the notion of principal type remains valid.

[3]The notion of a partially ordered set of sorts has been more fully developed and applied to other problems by Goguen and Meseguer [6].

$$Constr(\zeta') = \bigcap_{\zeta_i \in Z_C} Constr(\zeta_i)$$

as a unique least sort to which C belongs.

2.1 Example – the language TYPES

As an example of the natural types of an abstract grammar, consider the following definition, in METAL, of an abstract grammar for a language expressing type definitions themselves.

```
Phylla:
      TYPE_DEF   ::= type_def ;
      TYPENAME   ::= type_id type_inst ;
      TYPEID     ::= type_id ;
      UNION      ::= product constr_list ;
      APP        ::= constr constr_app ;
      CONSTR     ::= constr ;
      TYPELIST   ::= type_exps ;
      TYPE       ::= type_id product type_inst arrow var ;
Signature:
      type_def     -> TYPENAME UNION ;
      type_inst    -> TYPEID TYPELIST ;
      type_exps    -> LIST(TYPE) ;
      constr_list  -> LIST(APP) ;
      constr_app   -> APP TYPE ;
      product      -> LIST(TYPE) ;
      arrow        -> TYPE TYPE ;
      type_id      -> implemented as IDENTIFIER ;
      constr       -> implemented as IDENTIFIER ;
      var          -> implemented as IDENTIFIER ;
```

We shall only be concerned with the definitions of the phylla, or sorts, which also provide the sets of operators that belong to each sort. We see that the subset ordering gives as the irreflexive elements of the subsort relation

$$\texttt{CONSTR} \preceq \texttt{APP} \qquad \texttt{TYPEID} \preceq \texttt{TYPENAME} \preceq \texttt{TYPE}$$

but that in addition, the operator product belongs both to the sort UNION and to the sort TYPE, which are incomparable by $\preceq$. A new sort is inferred,

```
PRODUCT ::= product ;
```

and the subsort relation is extended by

PRODUCT $\preceq$ UNION PRODUCT $\preceq$ TYPE

After adding the new sort, each operator belongs to a unique least sort,

```
type_def    : TYPE_DEF
type_inst   : TYPENAME
type_exps   : TYPELIST
type_id     : TYPEID
product     : PRODUCT
constr      : CONSTR
constr_list : UNION
constr_app  : APP
arrow       : TYPE
var         : TYPE
```

The idea is then, that any expression formed of an application of one of these operators to argument expressions of admissible types can itself be typed in the least sort to which the operator belongs. It will in turn be admissible in any sort that follows its least sort in the subsort relation.

2.2 Programmer-defined types

In addition to the natural types inherited from an abstract grammar, TYPOL allows the introduction of types defined by declaration in a program. The form of such a declaration is a signature specification similar to those of functional programming languages such as ML, Hope or Miranda. A signature specification defines a type in terms of a sequence of constructor declarations.

A constructor is a functor symbol that can be applied to a list of arguments to produce a term. A constructor's declaration gives its name and the types of its arguments. These may include the type that is being defined, thus a type definition can be recursive. A constructor may also be nullary, in which case the constructor's name alone serves to declare it. If a list of arguments can be typed to correspond to the type signature given in a constructor's declaration, then a constructed term is said to be well-typed and has the type in which the constructor is declared.

The requirement that all constructors be declared is an important distinction between TYPOL and untyped Prolog, which allows functor symbols

of unspecified sorts. A type-checking system imposed upon Prolog must attempt to infer appropriate types for the functor symbols used in a program by collecting functors to form signatures of a common sort. This procedure can be computationally laborious. It is unnecessary when constructors are declared with types.

Declared types have the same status as do the natural types of a grammar. The constructors declared for a type are analogous to the operators of the free algebra defined by a grammar. The names of declared types must not coincide with the names of natural types and the set of constructors of a type must be disjoint from the constructors of any other declared type or the operators of the natural types. Thus declared types are incomparable with respect to the subsort relation.

A type scheme can also be declared, by abstracting type variables from the type name (and its declared constructors). For example, a type of polymorphic pairs can be declared by

```
define PROD(A,B) ::= pair(A,B);
```

and a new type of lists could be declared (although lists are a built-in type of TYPOL, with infix constructor syntax) by

```
define LIST(A) ::= nil | cons(A,LIST(A));
```

A typical use made of programmer declared types in TYPOL is to introduce data structures convenient for semantic definition.

There are a number of predefined, polymorphic type schemes in TYPOL, including list types and a disjoint set of product types with infix constructors ',', ':', '->', etc.

3 Types for rules

A TYPOL rule is a conclusion and a list of premisses, which may be empty. Both conclusions and premisses are given in the form of sequents, where a sequent is written as $s_1, \ldots, s_m \vdash t_1, \ldots, t_n$. Intuitively, one may think of a sequent as relating a context to a consequent. A sequent `C` $\vdash$ `E` can then be read as 'conclude `E` from the context `C`'. This is the reading that would be given in a sequent calculus, for instance. In fact, the turnstile symbol, $\vdash$, is just a convenient relation symbol. The context expression to the left of a turnstile can be an arbitrary term, as can the consequent to its right. In

the implementation of TYPOL, the turnstile is translated into a predicate symbol for Prolog.

A *rule set* in TYPOL is a named collection of rules defining a set of relations over the well-typed terms of a language. In a system with typed terms, an n-place relation is typed as an n-fold product of term types. In the syntax of TYPOL, any of the relations defined in a rule set can be represented using the turnstile symbol as the sole relational operator. The meaning of the turnstile is conveniently overloaded within each rule set. The overloading is resolved by the type given to the conclusion of each rule. Rules having conclusion types in common are identified as defining components of a common relation.

Each occurrence of a sequent can be labeled with the name of a rule set in which it is intended to match a rule. A sequent without annotation is intended to be matched in the same rule set in which it occurs. A labeled sequent must agree in type with one of the relations defined in the rule set corresponding to its label. This is what it means for a sequent to be well-typed. For a rule to be well-typed means that its conclusion is well-typed and that each sequent among its premisses is well typed in the set with whose name it is labeled (or in the set in which it occurs, in case it is unlabeled).

When an ill-typed program is rejected, the typing system helps us to find and eliminate programming errors such as using the wrong number of arguments to a functor or a relation symbol, interchanging arguments in an expression, or misspelling the name of a constructor. Since it is possible to determine all constructors belonging to a type, a type checking system could also warn of manifestly incomplete specifications, that is, of a rule set in which there is no rule whose conclusion can match some sequent that is well-typed for that rule set.

4 Type-correctness of a TYPOL program

We can give a set of rules for well-typed programs in TYPOL. It is easiest to give these rules in the formalism of Natural Semantics[8], which has previously been used to express the type systems of a functional subset of ML, and of a minimal functional language[2] similar to Milner's example, *Exp*. We first give rules for ground types, later generalizing to programmer-defined, polymorphic type schemes.

In the formulas that follow, the variable ζ stands for an arbitrary sort; τ for a zero-order type, that is, either a sort or a (finite) cartesian product of sorts, excluding functional types; η for an arbitrary type. A finite set of zero-order types is designated by Σ.

The subsort relation is generalized to zero-order types by adding the axiom:

$$\frac{\tau_1 \preceq \tau_1' \ \ \& \ \ \tau_2 \preceq \tau_2'}{\tau_1 \times \tau_2 \ \preceq \ \tau_1' \times \tau_2'}$$

and to arrow types by the axiom:

$$\frac{\tau_1 \preceq \tau_1' \ \ \& \ \ \tau_2 \preceq \tau_2'}{\tau_1 \rightarrow \tau_2 \ \preceq \ \tau_1' \rightarrow \tau_2'}$$

Note that the subsort relation, generalized to arrow types, is monotonic in both the first and second domain of the arrow. This would not be satisfactory if higher-order function types were allowed.

As notation for the extension of a context, the operator $\uplus$ is used to indicate unique union, i.e.

$$\frac{A \ \vdash \ C}{A \uplus \emptyset \ \vdash \ C}$$

$$\frac{\{x : \eta\} \cup (A \uplus B) \ \vdash C}{A \uplus (\{x : \eta\} \cup B) \ \vdash \ C} \quad \text{if } (x : \eta') \notin A$$

The rules defining the TYPOL type system can be grouped into sets that specify (a) the types of expressions, (b) the meanings of type and variable declarations, (c) a finite set of types for a (named) collection of rules. The turnstile symbol used in these rules is itself annotated to reduce the degree of its overloading. A turnstile without annotation relates a typed expression or sequent to a context.

- Types of expressions

[VAR-INST] $A \vdash x : \zeta' \quad$ if $(x : \zeta) \in A \ \& \ \zeta' \preceq \zeta$

[CONSTR-INST] $A \vdash x : \eta' \quad$ if $(x : \eta) \in A \ \& \ \eta' \preceq \eta$

[APP] $$\frac{A \vdash e_1 : \tau \to \zeta \ \& \ A \vdash e_2 : \tau}{A \vdash (e_1\, e_2) : \zeta}$$

[PAIR] $$\frac{A \vdash e_1 : \tau_1 \ \& \ A \vdash e_2 : \tau_2}{A \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

[SEQUENT] $$\frac{A \vdash e_1 : \tau_1 \ \& \ A \vdash e_2 : \tau_2}{A \vdash (e_1 \vdash e_2) : \tau_1 \times \tau_2}$$

- Declarations of variables and types

[VAR-0] $$\frac{A \uplus \{x : \tau\} \overset{rules}{\vdash} Rules : \Sigma}{A \overset{rules}{\vdash} \mathbf{var}\ x\,;\ Rules : \Sigma}$$ (with undeclared type)

[VAR-1] $$\frac{A \uplus \{x : \tau\} \overset{rules}{\vdash} Rules : \Sigma}{A \overset{rules}{\vdash} \mathbf{var}\ x : \tau\,;\ Rules : \Sigma}$$ (with declared type)

[TYP-DEC] $$\frac{A \overset{cdecls}{\vdash} Cdecls : T \ \& \ A \overset{rules}{\vdash} Rules : \Sigma}{A \overset{rules}{\vdash} \mathbf{type}\ T = Cdecls\,;\ Rules : \Sigma}$$

[C-EMPTY] $A \overset{cdecls}{\vdash} \emptyset_{cdecls} : \tau$

[C-LIST] $$\frac{A \overset{cdec}{\vdash} Cdec : \tau \ \& \ A \overset{cdecls}{\vdash} Cdecls : \tau}{A \overset{cdecls}{\vdash} Cdec \mid Cdecls : \tau}$$

[CONSTR-0] $A \uplus \{C : \eta\} \overset{cdec}{\vdash} C : \eta$

$$\text{[CONSTR-1]} \quad \frac{A \overset{cdec}{\vdash} C\ \tau_1 \ldots \tau_{n-1} : \tau_n \to \eta}{A \overset{cdec}{\vdash} C\ \tau_1 \ldots \tau_n : \eta}$$

- Type consistency of a rule

$$\text{[HYPS]} \quad \frac{A \vdash Hyp_1 : \tau' \ \&\ A \vdash Hyp_2 : \tau}{A \vdash (Hyp_1\ \&\ Hyp_2) : \tau}$$

$$\text{[CON]} \quad \frac{A \vdash Hyp : \tau' \ \&\ A \vdash Con : \tau}{A \overset{rule}{\vdash} (Hyp \mid Con) : \tau}$$

- Combining rules within a set

$$\text{[COMB-0]} \quad A \overset{rules}{\vdash} \emptyset_{rules} : \emptyset_{types}$$

$$\text{[COMB-1]} \quad \frac{A' \overset{rule}{\vdash} R : \tau \ \&\ A \overset{rules}{\vdash} Rules : \{\tau_1, \ldots, \tau_k, \ldots, \tau_n\}}{A \overset{rules}{\vdash} R;\ Rules : \{\tau_1, \ldots, \tau \sqcup \tau_k, \ldots, \tau_n\}}$$

if $\tau \sqcup \tau_k$ exists and where $A \subseteq A'$

where $\sqcup$ is the least upper bound with respect to the subsort relation. The condition $A \subseteq A'$ allows that the context for each rule in a set must be a consistent extension of the context for the entire set. This allows local variables in the context of each individual rule whose types are inferred rather than declared.

$$\text{[COMB-2]} \quad \frac{A \overset{rule}{\vdash} R : \tau \ \&\ A \overset{rules}{\vdash} Rules : \{\tau_1, \ldots, \tau_n\}}{A \overset{rules}{\vdash} R;\ Rules : \{\tau, \tau_1, \ldots, \tau_n\}}$$

if $\tau \sqcup \tau_k$ fails to exist for $1 \leq k \leq n$.

$$\text{[SET]} \quad \frac{A \overset{rules}{\vdash} Rules' : \Sigma' \ \&\ A \overset{rules}{\vdash} Rules : \Sigma}{A \overset{set}{\vdash} \textbf{set}\ \mathrm{I}\ Rules'\ \textbf{end}\ \mathrm{I};\ Rules : \Sigma}$$

where (**set** I : Σ') $\in A$

- Use of a rule from a set

$$\text{[USE]} \quad A \vdash \text{I}(Seq) : \tau \qquad \begin{array}{l}\text{if (\textbf{set} I} : \Sigma) \in A \\ \text{and } \tau' \in \Sigma \text{ and } \tau \preceq \tau'\end{array}$$

  in which I(Seq) is an I-labeled sequent.

The rule [VAR-INST] allows the types of variables to be instantiated to sorts. Rule [USE], on the other hand, allows a rule to be applied to a sequent whose type is a specialization of that derived for the rule itself.

4.1 Resolving overloading

At this point, the reader may realize that the rules for combining the types of rules within a set, [COMB-0, COMB-1, COMB-2], are not required by our notion of type correctness, since the type of a rule set is simply the set of types of its rules. However, it is important to keep in mind the intuitive meaning of types given for rules. In most programs, the rules in a set are not given independently of one another, but instead cover the cases required for structural induction on some type, or phyllum of a grammar. Knowledge of the types over which a set of rules assert a relation is important diagnostic information for the TYPOL programmer. It can be used to help verify that the rules given are consistent and complete, as well as that they are well-formed.

Additionally, the resolution of turnstile-overloading that is provided by typing can help an implementation to reduce the number of possibilities for rule application that must be tried. Only rules of a type compatible with a goal sequent need be tried. Knowledge of the type selected in an application of rule [USE] furnishes just the information required. This information is available when type-checking is performed, and can be utilized in compiling an implementation of rule application by rewriting.

4.2 Polymorphic typing

In the rules given above, we have assumed that constructors have concrete types and therefore have not yet properly accounted for programmer-defined polymorphic type schemes. The general form of a type scheme is

$[x_1, \ldots, x_n]\eta$, in which we use the notation $[x_1, \ldots, x_n]$ to designate universal quantification of type variables. A type (with no bound variables) is trivially a type scheme. A type scheme may be instantiated to a type by substituting types for all occurrences of its bound variables. We use the notation $\tau\{\tau'/x\}$ to signify the term gotten from τ by substituting τ' in place of every occurrence of x.

We add to the notational convention that σ stands for a zero-order type scheme, possible containing bound occurrences of type variables, and ω stands for an arbitrary type scheme. We also generalize the notation Σ to represent a set of zero-order type schemes.

It is also convenient to define a partial order to express the relation of a type scheme to its instances. We write $\omega \sqsubseteq \eta$ to express the proposition that

$$\omega = \eta \text{ or } \omega = [x_1, \ldots, x_n]\eta' \text{ and } \eta = \eta'\{\tau_1/x_1, \ldots, \tau_n/x_n\}$$

for some types $\tau_1, \ldots, \tau_n$.

A new rule must be added to allow a polymorphically-typed constructor to be instantiated:

[POLY-INST] $\quad A \vdash x : \tau \quad$ if $(x : \sigma) \in A \;\&\; \sigma \sqsubseteq \tau$

Substitute for rules [VAR-1, TYP-DEC, C-EMPTY, C-LIST, CONSTR-0, CONSTR-1, SET, USE] the new rules given below.

$$\text{[VAR-1']} \quad \frac{A \uplus \{x : \sigma\} \overset{rules}{\vdash} \mathit{Rules} : \Sigma}{A \overset{rules}{\vdash} \mathbf{var}\ x : \sigma;\ \mathit{Rules} : \Sigma}$$

Note that in the rules for type declaration, the list of type variables occurring in a type scheme is carried in each rule. These variables are then used to construct appropriate type schemes for constructors of the type scheme (rule [CONSTR-0']).

$$\text{[TYP-DEC']} \quad \frac{A \overset{cdecls}{\vdash} \{x_1, \ldots, x_n\}, \mathit{Cdecls} : T\ x_1 \ldots x_n \;\&\; A \overset{rules}{\vdash} \mathit{Rules} : \Sigma}{A \overset{rules}{\vdash} \mathbf{type}\ T\ x_1 \ldots x_n = \mathit{Cdecls};\ \mathit{Rules} : \Sigma}$$

[C-EMPTY'] $\quad A \overset{cdecls}{\vdash} \{x_1, \ldots, x_n\}, \emptyset_{cdecls} : \sigma$

$$\text{[C-LIST']} \quad \frac{A \overset{cdec}{\vdash} \{x_1,\ldots,x_n\},\ Cdec : \sigma \ \ \&\ \ A \overset{cdecls}{\vdash} \{x_1,\ldots,x_n\},\ Cdecls : \sigma}{A \overset{cdecls}{\vdash} \{x_1,\ldots,x_n\},\ Cdec \mid Cdecls : \sigma}$$

$$\text{[CONSTR-0']} \quad A \uplus \{C : [x_1,\ldots,x_n]\tau\} \overset{cdec}{\vdash} \{x_1,\ldots,x_n\},\ C : \tau$$

$$\text{[CONSTR-1']} \quad \frac{A \overset{cdec}{\vdash} \{x_1,\ldots,x_n\},\ C\ \tau_1 \ldots \tau_{n-1} : \tau_n \to \tau}{A \overset{cdec}{\vdash} \{x_1,\ldots,x_n\},\ C\ \tau_1 \ldots \tau_n : \tau}$$

The set of types of a rule set is generalized to a set of type schemes by quantification of the unbound, free type variables occurring in each type.

$$\text{[SET']} \quad \frac{A \overset{rules}{\vdash} Rules' : \Sigma' \ \ \&\ \ A \overset{rules}{\vdash} Rules : \Sigma}{A \overset{set}{\vdash} \textbf{set}\ \mathrm{I}\ Rules'\ \textbf{end}\ \mathrm{I};\ Rules : \Sigma}$$

where $(\textbf{set}\ \mathrm{I} : \Sigma'') \in A$ and $\Sigma'' = \{Gen_A(\tau) \mid \tau \in \Sigma'\}$

in which the generalization of a type relative to a context is defined as:

$$Gen_A(\tau) = [x_1,\ldots,x_n]\tau, \qquad \{x_1,\ldots,x_n\} = FV(\tau) \setminus FV(A)$$

and $FV(t)$ is the set of free variables of a t, type or an environment.

$$\text{[USE']} \quad A \vdash \mathrm{I}(Seq) : \tau \qquad \begin{array}{l}\text{if } (\textbf{set}\ \mathrm{I} : \Sigma) \in A \\ \text{and } \sigma \in \Sigma \text{ and } \sigma \sqsubseteq \tau' \text{ and } \tau \preceq \tau'\end{array}$$

5 Implementing type inference

Since type inference is intended to be available for use in an interactive environment, it is crucial that the algorithms for performing inference should be quite fast. The key to an inference algorithm for a polymorphic type system is the unification of a pair of type expressions, which attempts to find a most general demonstration of the proposition that the expressions have common instances. Ordinary unification would be adequate for the type system we have specified were it not for the subsort relation. Rules [VAR-INST, CONSTR-INST, USE] may require specialization of a sort and rules [COMB-1 COMB-2] require generalization to the least upper bound

of a pair of sorts. The operation $\sqcup$ can be implemented as a unification, but one that takes account of the subsort relation as well as instantiating variables. Otherwise, the $\sqcup$ relation might be expressed by rules, in which case, search for a required generalization would occur by first trying to match a particular sort, then repeatedly applying the rules for the subsort relation in order to generalize whenever a match initially failed. In order to obtain efficiency, the operator $\sqcup$ should instead be implemented directly by a unification algorithm extended with the theory of order on subsorts[10].

Acknowledgements

Thanks are due to Thierry Despeyroux and Gilles Kahn for their explanations, encouragement and review of early versions of this report.

References

[1] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual, Centaur: the system, INRIA Res. Rept. No. 777, Dec. 1987

[2] D. Clement, J. Despeyroux, T. Despeyroux, G Kahn, A simple applicative language: Mini-ML, INRIA research report No. 529, May, 1986.

[3] T. Despeyroux, TYPOL: A formalism to implement natural semantics, in *CENTAUR Version 0.5 Programmer's Manual*, INRIA, Sophia-Antipolis, Feb., 1988.

[4] R. Dietrich and F. Hagl, A polymorphic type system with subtypes for Prolog, in *ESOP '88*, H. Ganzinger (ed.), Springer-Verlag LNCS vol. 300, pp. 79-93, 1988.

[5] Y.-C. Fuh and P. Mishra, Type inference and subtypes, in *ESOP '88*, H. Ganzinger (ed.), Springer-Verlag LNCS vol. 300, pp. 94-114, 1988.

[6] J. A. Goguen and J. Meseguer, Order-sorted algebra solves the constructor-selector, multiple representation problem, *Proc. of Second LICS Conf.*, IEEE, 1987.

[7] R. Hindley, The principal type-scheme of an object in combinatory logic, *Trans. Amer. Math. Soc. 146*, No. 1, pp. 29-60, 1969.

[8] G. Kahn, Natural semantics, INRIA research report No. 601, Feb., 1987.

[9] R. Milner, A theory of type polymorphism in programming, *Jour. Comp. & Sys. Sci. 17*, pp. 348-375, 1978.

[10] J. Meseguer, J. A. Goguen and G. Smolka, Order-sorted unification, *Proc. Colloq. on Resolution of Equations in Algebraic Structures*, 1987.

[11] A. Mycroft and R. A. O'Keefe, Polymorphic type system for Prolog, *Artificial Intelligence 23*, pp. 295-307, 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique